1983-06

A study of programmer productivity metrics for fleet material support office (FMSO).

Hughes, Gary Jack

Dudley
Monter

Dudle
Monte

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A STUDY OF PROGRAMMER PRODUCTIVITY METRICS
FOR FLEET MATERIAL SUPPORT OFFICE (FMSO)

by

Gary Jack Hughes

June 1983

Thesis Advisor: Dan Boger

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Study of Programmer Productivity Metrics for Fleet Material Support Office (FMSO)	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1983
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Gary Jack Hughes	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: June 1983
	13. NUMBER OF PAGES: 93
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programmer Productivity, Productivity Metrics, Productivity, Programmer Production Function, Performance Metrics, Programmer Productivity Metric Comparison

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The demand for software programs is increasing at an ever faster pace than supply. As a result, software has become the most expensive part of a computer system's life cycle costs. Accordingly, software development efficiency has become a major managerial concern. This paper discusses the software development process within the context of the production function. It presents a comparison of various productivity models that are currently being discussed in the literature and a test of selected models. This paper is part of a group of papers which together provide

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Dudl
Mont

recommendations to the Fleet Material Support Office (FMSO) to enhance its software development organization.

Approved for public release; distribution unlimited

A Study of Programmer Productivity Metrics for Fleet Material Support Office (FMSO)

by

Gary Jack Hughes
Lieutenant Commander, United States Navy
B.A., Pacific University, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

ABSTRACT

The demand for software programs is increasing at an ever faster pace than supply. As a result, software has become the most expensive part of a computer system's life cycle costs. Accordingly, software development efficiency has become a major managerial concern. This paper discusses the software development process within the context of the production function. It presents a comparison of various productivity models that are currently being discussed in the literature and a test of selected models. This paper is part of a group of papers which together provide recommendations to the Fleet Material Support Office (FMSO) to enhance its software development organization.

Dudl
Mont

TABLE OF CONTENTS

I. INTRODUCTION 8

II. BACKGROUND 13

A. PRODUCTION FUNCTION 13

B. PRODUCTIVITY MEASUREMENT 16

C. MEASUREMENT PROBLEMS 17

D. MANAGEMENT AND PRODUCTIVITY MEASURES 20

E. MANAGEMENT'S PROBLEM 21

III. METRIC COMPARISON 24

A. LINES OF CODE 26

B. PROGRAM FUNCTION 35

C. USER FUNCTIONS 36

D. MODEL RECOMMENDATIONS 37

IV. METRIC TEST 39

A. DEFINITIONS 39

B. TEST PROCEDURES (APPLICATION SOFTWARE) 41

C. TEST RESULTS (APPLICATION SOFTWARE) 44

D. TEST PROCEDURES (MAINTENANCE) 51

E. TEST RESULTS (MAINTENANCE) 53

F. MANAGEMENT CONSIDERATIONS 54

V. PRODUCTIVITY PERSPECTIVES 57

A. MANAGEMENT 58

B. ENVIRONMENT 62

Dudle
Monte

C. PEOPLE 62

D. PROCESS 66

E. IMPROVEMENT PROJECTIONS 70

VI. CONCLUSION AND RECOMMENDATIONS 72

APPENDIX A: ALBRECHT'S DEFINITIONS AND WORKSHEET 74

APPENDIX B: RRMIS DATA 76

APPENDIX C: BOEHM'S MOEDL DATA 77

APPENDIX D: JOHNSON'S MODEL DATA 79

APPENDIX E: ALBRECHT'S MODEL DATA 80

APPENDIX F: CONFIDENCE INTERVALS (JOHNSON'S MODEL) 81

APPENDIX G: CONFIDENCE INTERVALS (ALBRECHT'S MODEL) 82

APPENDIX H: MISIL MAINTENANCE DATA 83

APPENDIX I: JOHNSON'S MODEL DATA (MAINTENANCE) 84

LIST OF REFERENCES 85

BIBLIOGRAPHY 88

INITIAL DISTRIBUTION LIST 92

Dudle
Monte

LIST OF FIGURES

3.1 Problems With Lines of Code 28

4.1 Estimated to Actual Productivity Regression (Boehm's Model) 45

4.2 Lines of Code to Program Size Regression 47

4.3 Function Points to Program Size Regression 49

4.4 Function Points to Program Size Regression 50

5.1 Range of Individual Differences in Programming Performance 63

5.2 Ranking of Programming Performance on Five Objectives 65

Dudle
Monte

I. INTRODUCTION

The increasing use and complexity of computers coupled with the rising costs of programmers has created a situation that now demands management attention be focused upon computer and programmer performance. This is true in both the public and private sector. Scrutiny is now directed in several areas. First, computer centers, historically an overhead cost, must now show their worth and compete head-to-head with other organizational endeavors for scarce operating and investment dollars. Their costs must be compared against their returned value and their cost/benefit ratios must be compared against organizational and industry-wide standards. These comparisons give management an indication of and a perspective on in-house performance. In essence, they tell management if they are getting their money's worth. A second related area of managerial attention has been in the installation of recommended improvements. Because the expenditures in software development are so large, lower management must now closely monitor the benefits of the improvement to make sure they are actually received. No longer will upper management allow new equipment justifications to end at decision time. They must now weather the test of reality. They must return the expected

Dudle
Monte

benefits. A final area of concern is the schedule. Traditionally, cost over runs on computer projects were expected, even planned for by management. However, as costs continue to climb, the tolerance is diminishing. Management must now control the development effort to a greater extent. They must know if they are on schedule, and if not, what the ramifications are. To do this, management must know how long the development time is, how much it will cost and the critical path. All these managerial concerns require quality and quantity measures to be made on program and programmer performance. Productivity must be measured and put into proper perspective if an organization is going to compete in today's environment.

Reflective of this changing environment has been the edicts of Congress. The Federal Register of 5 April 1979 [Ref. 1] highlights this new emphasis in its discussion of revised Federal policy concerning the acquisition of commercial and industrial products and services (change to OMB Circular A-76). Simply stated, where possible, the government's policy is now one of reliance on the private sector for goods and services, giving proper attention to relative costs. What this means to the manager of a government-owned and run general purpose computer center or software house is literally direct competition with their private counterparts. In fact, they both submit bids on the

Dudle
Monte

work. If the private firm can produce the same product for less money they will be awarded the contract. For both managers, submission of the bid requires knowing what resources are consumed in the production of a specific project. All inputs must be considered. In software development the primary input is programming effort. Therefore, the productivity of the programmers must be known. No longer can government software houses guess at their productivity. They must know precisely or they will be out of business.

Another concern for the government manager is the programmers themselves. The demand for programmer services is much greater than the supply and the situation is getting worse. According to James Martin, this is "the most serious constraint slowing down effective use of computers..." [Ref. 2]. In order to compete effectively, a manager of an ADP center must get the best programmers available. This requires individual programmer performance measurement and evaluation.

A second area of Congressional focus has been on governmental ADP managers and their control of software conversions during system upgrades. Historically, the performance has been terrible [Ref. 3]. Cost over runs, later followed by conversion failures, have typified the government's record. Erroneous productivity projections were

Dudle
Monte

often found to be one of the major underlying problems. The amount of effort and expertise required to do the conversion was under estimated and the productivity of the workers assigned over estimated. Accurate productivity projections would have resulted in better planning estimates and in tighter control of the conversion process. Managers of computer centers and software houses must realize that productivity measurements are critical, even essential, to control of the software development effort.

All of the above problems and concerns apply to the Fleet Material Support Office (FMSO). As a mass producer of general purpose software programs (approximately 8000) they face competition from the private sector for the work they perform. During The development phase of individual programs, they face schedule and cost problems that must be controlled. As the employer of several hundred programmers, they face a monumental task of performance rating and evaluation. For these reasons, programmer productivity must be measured at FMSO.

An insight into how FMSO can come to grips with the above discussed problems is the purpose of this paper. As a first step, productivity in a generic sense will be addressed within the framework of the production function. With that background as a foundation, the various programmer productivity metrics currently found in the literature will

Dudle
Monte

be reviewed, analyzed, and compared. Selected models will be tested using FMSO data in order to measure their predictive abilities. Conclusions and recommendations will be drawn and presented for FMSO's consideration and adoption based upon the results of the test.

Dudle
Monte

II. BACKGROUND

Generally speaking, productivity is defined as the relationship between the volume of services or goods produced and the physical inputs required in their production. It is a ratio of output divided by input. Since it is a time sensitive measurement, comparison of two or more ratios can reveal characteristics to the software manager that are of major managerial concern. Productivity decline, stability and growth trends, and efficiency measures can be important indicators to management of long and short range organizational well being. They can identify levels within the organizaton that need attention and the consequences of change. For these and other reasons, it is imperative that the software manager understand the underpinning concepts that support productivity measurements.

A. PRODUCTION FUNCTION

Underlying the input-to-output relationship is the concept of the production function, which is the notion that the physical unit of output is dependent upon the inputs used in the production process and the efficiency in which they are utilized. The inputs are normally categorized into three classifications: labor, capital and materials. There are two types of productivity ratios used: total and partial

Dudl
Monte

productivity ratios. A total factor productivity ratio includes all of the inputs while a partial factor productivity ratio usually addresses a single input. Inputs that are consumed in the production process are considered consolidated and consigned to the produced output. The degree of consumption provides an indication as to the efficiency of the overall function.

The measured output in the production function should always be a final and not an intermediate product. As an example, the output measured for the production function of a farm should be the food actually produced and not the acreage planted. The amount each acre produces is a single process within the function (very disaggregated product) and the use of this value as a productivity indicator can be misleading.

Another characteristic of a production function is that given quantities of output can usually be produced with differing combinations of inputs. Less water and more fertilizer can produce the same amount of food. An optimal combination of the inputs will provide a least cost solution for producing outputs with the same marginal value. There is, then, a point where the amount of additional fertilizer necessary to replace a unit of water will cost more than the amount saved. Consequently, the mix of inputs is considered optimal when their marginal cost/value ratios are equal. Therefore, the ratio of the cost of the water used to what it

Dudle
Monte

will produce is equal to the ratio of the cost of fertilizer used and what it will produce.

Over time the optimal mix of inputs is not stable. According to Kendrick, it will alter as a result of changing relative input prices, increasing technical knowledge or differing quantities of received output (if returns to scale are not constant) [Ref. 4]. Since the optimal mix changes, ratios of output to a single input (partial factor productivity ratio) should not be used to measure and compare productivity efficiency. Therefore, food produced per man-hour of labor should not be used as a measurement of productivity for the farm example. This can be a misleading measurement for several reasons. First, labor can be substituted for or by other inputs (non-optimal solution). Secondly, this tradeoff may affect the influence other inputs have on the output. Finally, changes in the efficiency of the production function can affect such measurements. The use of total factor productivity ratios allows input categories, such as labor, to be further broken down (skill level, work type, etc.), which in turn can facilitate better managerial analysis and problem identification. As a rule, inputs should be specifically identified if their physical characteristics and/or prices differ substantially from other inputs.

Dudle
Monte

B. PRODUCTIVITY MEASUREMENT

Changes in productivity are determined through comparison, either with other productivity measures or historical data. Total factor productivity changes are often defined in the literature as changes in production efficiency. These changes may be the result of changing technology, changes in the scale of output and/or changes in the rate of utilization of capacity [Ref. 5]. New innovations in technology allow more efficient conversion of inputs to outputs. Managerial decisions that cause changes in the volume of output can bring about efficiency improvements which are explained within the principles of economies of scale. Additionally, changes in the rate and mixture of inputs will cause more or less efficient use of those inputs. All of these factors alter the efficiency of the production function accordingly. Changes in productivity can be influenced by both short and long range factors. In the short run, changes in output capacity requirements will directly affect the productivity ratio because of the somewhat fixed nature of the inputs. The number of people employed, the materials currently on order or stocked for production and the physical plant form temporary constraints on the production function. These constraints take time to change. Daily fluctuations in output demand must work within their confines. Additional elements that can cause short run

Dudl
Mont

productivity changes are learning curve factors as employees adapt to physical and organizational change. Long run changes in productivity can be attributed to the changing quality of the inputs over time or by managerial initiatives, such as decisions which bring about changes in the scale of output [Ref. 6]. Some short range factors, such as organizational change, may cause temporary losses of productivity yet ultimately result in long term gains. Collected historical data on the degree of influence that particular changes have on the production function can be a valuable managerial decision making tool.

C. MEASUREMENT PROBLEMS

One of the major problems in measuring productivity is the lack of a single concept of efficiency. This unclear definition is partly attributable to the multidimensionality of the inputs and outputs. It is often difficult to determine what is and is not an input or output. Equally confusing can be the categorization of the inputs into heterogeneous atomic elements. The labor input illustrates this problem. Labor can be broken down into many different job types and skill levels. Inclusion of all the subunits is impractical and probably impossible. Management must, therefore, decide what segregations and aggregations of inputs are to be used because it will affect later analysis. In that regard, management will have to distinguish between

Dudl
Mont

core and peripheral inputs; and within the peripheral inputs, those to be included in the productivity ratio. For example, should the electricity being used to light the offices and the production plant be included (or pro rated) as input to the production process? These and other problems of measurement must be decided by management.

A second problem in measuring productivity is the changing nature of the inputs and outputs over time. Quality changes in the inputs can affect the production function's efficiency or the output quantity. Conversely, changes in the quality of the outputs suggest changes in the production function or in the input quality. Differences of quality in inputs and outputs are hard to detect. When detected, they are hard to measure. When measured, their influences are hard to determine. The elusiveness of this variable complicates the compariability of productivity ratios through time.

A third problem is imprecise productivity measuring tools. Often the input or output simply cannot be measured by standard means. Its value is abstract and difficult to determine. This is often the case in the public sector (service-oriented organizations), ie., police departments, politicians, etc.. When inputs and outputs cannot accurately be measured, and management substitutes measurable intermediate outputs as indicators by which to judge

Dudl
Mont

performance, gaming can occur. An example of gaming can be found when secretaries are evaluated on the number of pages they type. The percentage of "white space" on the typed documents will increase as they try to type less on more pages.

Confusion of technical efficiency with economic efficiency is another problem. Given output levels can be produced using various input mixes. The selected input mix is considered technically efficient if it minimizes the input requirements. For example, if a dam, requires as a minimum ten laborers with heavy equipment for construction, the use of eleven laborers with heavy equipment is a feasible yet technically inefficient solution. For a given output level, there can be numerous solutions which are technically feasible. A dam can be constructed with either ten laborers using heavy equipment or with one thousand laborers using shovels. In between these two ends of the spectrum there are countless other technically efficient solutions. A line connecting the locus of all the technically efficient methods for producing a given level of output forms an isoquant. At a point(s) along the isoquant the solution that is the least cost and, therefore, most economically efficient can be found. This point is determined by comparing solution mix input costs. The mix that provides economic efficiency may not be the same over time or at all locations. In

Dudl
Mont

undeveloped countries where there is an abundance of labor and a scarcity of capital, the use of one thousand laborers with shovels may be a technically and economically efficient solution. Conversely, in industrial countries less labor and more capital may provide the optimum solution.

Other problems concern the confusion of productivity with production or capacity measurement. Productivity measures the efficiency by which resources are used and not the degree of utilization of the available resources. For this reason, productivity measurements should not singularly be used to determine/justify increases in employee compensation.

D. MANAGEMENT AND PRODUCTIVITY MEASURES

Productivity measures serve three main puposes of management: planning, control (decision making) and evaluation. Historically, recorded productivity data on past or similar projects has provided the basis by which future requirements are determined. This base, modified as necessary to reflect new constraints, is used to establish new or recurring short and long term project goals and objectives. Additionally, it helps to identify resource timing requirements within those projects.

Managerial monitoring of short term goal attainment is accomplished via budget and scheduling measurements using various productivity metrics. Three tracking techniques are predominately in use. The first is a comparison of actual to

Dudl
Mont

planned expenditures. Identified variances between project estimates and actual disbursements can indicate improper financial management or inaccurate program projections. Secondly, work accomplished can be measured against work scheduled. Several models exist, such as CPM, PERT, etc., which help management not only monitor work accomplishment but also identify the critical path, areas of slack, and probabilities of milestone attainment. Finally, a third comparison can be made between budget and schedule variances. The differences between actual and planned expenditures is compared against the differences between actual and scheduled work accomplishment. Variance relationships between the two can be a powerful indicator of organizational health to management [Ref. 7].

The evaluation phase measures how well the organization is meeting its long term objectives. Using productivity measures, areas of improvement and deficiency can be identified and analyzed. Resulting data can then be used to recalibrate planning models and update baselines.

E. MANAGEMENT'S PROBLEM

Generally speaking, productivity is the relationship of a unit of output to its required inputs. This relationship is based upon the concept of a production function, where inputs are received and processed in the production of output. In order to accurately measure productivity (within the

Dudl
Mont

production function) all of the inputs (capital, labor and material) required to produce the final output must be considered. In this regard, there are several characteristics of the production function that can change or be controlled and, therefore, are of managerial concern. First, a given output can be produced with different input mixes. Secondly, of the various input mixes possible, usually only one is optimal. Finally, the optimal mix is not stable and will fluctuate over time.

Further complicating management's productivity measurement effort are several problems with the actual measurement. First, since it is a relative measurement, it requires a comparison to be made with either accepted standards or historical data. Secondly, it is often hard for management to define exactly what inputs and outputs should be included in the measure. This determination is made more difficult by the fact that inputs and outputs change over time and that there are only imprecise measurement tools currently available. A final problem is confusion with the term efficiency. It is possible to be technically but not economically efficient. Management must be aware of the difference, what it means and how to correct it.

Despite the above discussed difficulties of definition and measurement, it is essential management successfully measure and track productivity because it is singularly the

most important indicator of corporate performance. Management must know where it presently is at and what changes must be made to meet organizational goals and objectives. These measurement requirements create many non-trival problems. Accordingly, management must be aware of the pitfalls and ask the appropriate questions in order to ensure meaningful answers. The difficulty of this task, as it relates to programmer productivity, will be demonstrated in the next chapter.

III. METRIC COMPARISON

Within the framework developed in Chapter Two, the area of programmer productivity can now be discussed. First, however, terms must be defined. Programmer productivity is often described as the quantity of work produced in a unit of time. In order to better understand this definition, the inputs and outputs of the production function need to be examined. A partial listing of the inputs used in producing software are computer time (capital), software tools (capital), computer terminals (capital), software programmers (labor), clerical help (labor), management (labor) and consumables (material). Changes in the quality of any of the inputs, such as programmer skill level or software tools, will accordingly affect the output. Useable code is considered the output of the programmer production function, which is part of the problem in trying to measure programmer productivity. Useable code is hard to define, and harder yet to measure. It is to some extent a quality judgement. Still, certain characteristics of useable software are known. First, it must satisfy the user. Additionally, it should incorporate several fundamental software concepts such as modifiability, efficiency, reliability and understandability [Ref. 8].

Programmer productivity when viewed within the background of the production function highlights several misconceptions and misunderstandings that abound in the literature. First, programming is an input to and not an output from the production function. It is not an end in itself. Conversely, the programming effort in conjunction with the other inputs produce computer code, an intermediate product. When executed, the code provides the customer with a useable final product. This sequence of events underscores a second common misunderstanding, namely, that code is the final output of the programming effort. Code is an innate object, with user value borne in execution. The code's useability, not length, determines its value. The difference in the length or characteristics of the source or object code is transparent to the user. Only the value of the delivered results are important. These simple concepts are consistently blurred in the literature. On balance, this confusion and lack of a clear notion of the programming productivity function is a major contributing factor to many other problems that plague programmer productivity measurement.

The most common programmer productivity metrics found in the literature fall into three general classification groups: (1) lines of code, and functions which the (2) program and (3) user perform. Each of these areas can be usefully

viewed as a process in the production function. The processes' exact relationship within the function can be an indication of how good the metric can measure productivity. Therefore, a discussion of the predominant models in each of the three categories will be presented and evaluated with respect to the production function. The presentation will consist of a brief description of the model, its popularity of use, and the inputs it utilizes. Additionally, the evaluation will address some of the advantages and disadvanbtages of the models.

A. LINES OF CODE

The most common form of measuring programmer productivity discussed in the literature is lines of code. It is the predominant model because of its simplicity. Lines of code are easily counted. A line of code usually refers to the eighty character line that is used in coding programs, even though less than the full eighty characters are normally used. It is a source statement. The number of lines produced divided by the time expended in their production forms the ratio that is most often used for the measurement. Lines of code per programmer day or month are the most common ratios.

Programmer code, as discussed earlier, is an intermediate product in the production function. It is an output of one process and an input to another within the function. As

such, its use as a productivity indicator may be misleading. Additionally, lines of code per time unit is a partial factor productivity ratio, and this causes problems as discussed in Chapter Two. Lines of code is not the only input into the process, other inputs also exist. Results received from using a lines of code measurement should be tempered with an understanding of the model's limitations.

Several problems exist with lines of code measurement. First, what actually constitutes a line of code is unclear. One author listed fifteen different active definitions of what can be counted as a line of code. These variations are listed in Figure 3.1. Between the extremes, it is possible to have more than a two to one variance on a lines of code count for the same program. The problem is not, however, a critical one. For an individual company developing its own metric, the definition of what a line of code is must simply be stated at the outset. For organizations that intend to use an established model, the correct definition needs to be determined and applied.

A second problem involving the use of lines of code is the way it tends to penalize the use of high-level languages. Programs written in lower-level languages, such as assembler language, normally require more lines of code, as compared to higher level languages, in order to produce the same output. Using a line of code ratio as a measurement, the results

VARYING DEGREES OF "LINES OF CODE"

1. ONLY EXECUTABLE LINES
2. EXECUTABLE LINES AND DATA DEFINITIONS
3. EXECUTABLE LINES AND DATA DEFINITIONS AND COMMENTS
4. EXECUTABLE LINES AND DATA DEFINITIONS AND COMMENTS AND JCL
5. DELIVERED LINES ONLY
6. DELIVERED LINES AND SUPPORT SOFTWARE
7. DELIVERED LINES AND SUPPORT SOFTWARE AND TEST CASES
8. DELIVERED LINES AND SUPPORT SOFRWARE AND TEST CASE ABD SCAFFOLD CODE
9. NEW LINES ONLY
10. NEW LINES AND CHANGED LINES
11. NEW LINES AND CHANGED LINES AND RESIDUAL LINES
12. MACROS COUNTED ONCE (OR INCLUDED CODE)
13. MACROS COUNTED ON EACH USAGE (OR INCLUDED CODE)
14. "LINE" MEANING A PHYSICAL LINE ON A CODING PAD
15. "LINE" MEANING STATEMENTS BETWEEN DELIMITERS

SOURCE: Jones, T.C., "The State of the Art of Software Development," ACM Professional Development Seminar, College Park, MD, 7 April 1981.

Figure 3.1 Problems With Lines of Code Measurement

could indicate that the primitive or lower-level language is more productive. This is obviously not the case. Discrepancies such as this are indicative of the problems that can arise when intermediate and not final outputs are used to measure and judge productivity performance.

An additional problem with using a lines of code measurement is that it implies that the coding of the program is the most important part of the software development cycle. This is often not the case. The misdirection of emphasis is partly attributable to the use of a partial productivity measurement as the measuring tool. While highlighting the code writing effort, it overshadows the importance of the other inputs. As a result, noncoding tasks are often not measured. This omission can cause ridiculous results from the metric. T. C. Jones pointed out the paradox of the problem as follows [Ref. 9]:

> With modern defect prevention and defect removal techniques in programming, it sometimes happens that no defects are discovered during testing because the program has no defects at the time the test is carried out. If testing is done by an independent group rather than by the programmers themselves this tends to introduce slack time into development. By normal program development practice, the programmer usually cannot be fully reassigned until testing is over, in case defects should be discovered. Since it is nonproductive, slack time does not contribute to lines of code per programmer-month. It is therefore inaccurate to say for example, that one's productivity is one thousand lines of code per month during testing when there is no coding, and much of the time is spent waiting for bugs that may never occur. It is reasonable to say that slack time has added one month to a project but it is not reasonable to say that slack has proceeded at a rate of one thousand lines of code per month.

Duc
Mor

There is, however, a simple solution to this problem. During the slack period either assign the involved programmers other work (administrative/new project) or do not count the time.

A fourth problem with the use of lines of code is its awkwardness when aggregating independent measures of parts of the programming development cycle. Because of the measure's structure, it is easy to fall into the trap of double and triple counting the number of lines of code produced. The point is best demonstrated with the following example [Ref. 10]:

> Suppose a program consisting of 1000 lines of source code has been developed. The development cycle consists of four separate activities, each of which has taken one month to complete and has yielded a total development expenditure of four programmer-months. The sum of four consecutive activities, each of which proceeded at a rate of 1000 lines of code per month, is not 4000 lines of code per month, but 250 lines of code per programmer-month.

A fifth problem with lines of code measurement is that it does not adequately deal with quality differences. This deficiency is understandable since lines of code is an intermediate product with no user interface (quality is determined through usage). For example, succinctness is penalized. If two programs are similar in language and delivered results, the metric will indicate that the programmer which uses more lines of code is more productive. In fact, the opposite is true. The programmer with the fewest lines of code will produce better code because it will

use less of the other inputs (ie., cpu cycles, etc.) in execution. For this reason, lines of code measurement is extremely susceptible to gaming.

Currently, there is no proven solution to the quality measurement problem; however, several interesting theories do exist. The most promising quality measures are the works of Halstead [Ref. 11] and McCabe [Ref. 12]. Halstead's hypothesis simply states that the count of operators and operands contained in a program can be used to measure the complexity, predict the length and estimate the effort required to generate a specific program or algorithm. In brief, Halstead's metrics try to scientifically measure the psychological complexity of the program. McCabe's software complexity model is based on the number of basic control pathways that the software contains. It is a measure of the computational complexity of the program. It is also an attempt to develop a mathmatical measurement model for software productivity. For both models, a theoretical assumption, based on empirical data, is that an inverse relationship between complexity and quality exists. Presently, the literature indicates that neither model adequately measures software quality to the extent necessary; however, the research is encouraging [Refs. 13,14,15,16].

The use of a ratio to measure productivity also presents a problem. Implicit within the use of ratios is the natural

provided for estimating the number of man months of effort required to develop a software program in terms of thousands of deliverable source instructions. A second equation estimates the development schedule in months. Productivity for a specific program is estimated by dividing the initial user estimate of program size by the effort estimator. Basic COCOMO can be used to quickly develop a rough estimate of the software development costs. In the more advanced versions, subjective software product, computer, personnel, and project attribute multipliers are used to tune the model for more accurate performance. This is attractive because it allows a company to start with a simple metric and build from there. One unique advantage of this model is its ability to measure productivity in software maintenance activities. Although very little appears in the literature to indicate how well the model perfoms, it is believed that the COCOMO model can provide a reasonably good starting point for measuring productivity.

Another model for measuring programmer productivity was suggested by Walston and Felix [Ref. 18]. The model calculates programmer productivity as the ratio of delivered source lines of code to the total effort (man-months) required to complete the given program. Five major parameters: productivity, schedule, cost, quality, and size (listed in order of increasing difficulty and complexity of

analysis) were identified that significantly influence productivity estimates. Additionally, twenty-nine independent variables were identified in these categories to be significantly correlated with measuring programmer productivity. The combined variables form a productivity index. Felix and Walston's model has received some criticism in the literature on two points. First, many of the variables require subjective measurements, ie., the degree of user participation in the definition requirements. To an extent, the criticism also applies to Boehm's model. Secondly, the data base on which the model is based was collected on a project rather than a program basis. There is fear that the projects' long duration may have unmeasurably influenced the isolated variables [Ref. 19].

James Johnson suggested a third model for measuring productivity [Ref. 20]. The model is a data base comparison using historical lines of code counts (comments and all other statements are counted as lines of code) for similar projects. The counts were obtained from automatic librarian statistics and estimates. Man-day figures used included both productive and nonproductive time. Averages for lines of code per hour for small and large programs were then determined, along with the variances. These figures in a general way, are used as a measure of productivity. Subjective opinion was used to estimate technology levels,

difficulty and staff quality. It was concluded by Johnson that lines of code averages can be used at a macro level for project estimating. Although a simplistic approach to the problem, as compared to the other models, this metric can have useful application as a rough indicator of performance.

B. PROGRAM FUNCTION

A productivity metric has been suggested that uses man-hours per function as a measure [Ref. 21]. Functions are defined as a section of program that performs only one activity, has only one entry and exit point, employs the logic principles of structured programming and has between five and fifty source statements. The functions of individual programs are counted and then divided into the respective man-hours spent on development. The resulting ratios are then compared against an existing data base in order to determine performance.

Functions, like line of code, are an intermediate product within the production function. As a result, many of the problems that lines of code have also apply to function measurement. One new problem is function definition. In a structured format, a function normally means a paragraph. However, the definition can also be construed to mean subroutine, procedure, etc. What should be counted is unclear. This confusion can result in gaming to the extent programmers can control program structure. Like lines of

code, software quality is not measured in program functions. This is probably the major deficiency of the model.

Trevor Crossman, the metric's principle proponent, discovered for the six projects tested that the man-hours per function ratios clustered around the values of two and four [Ref. 22]. He also determined the functions that required approximately four man-hours per function to complete were for new or "breakthrough" technology. A learning curve was involved. Other variables were tested and found not to influence the ratio. Crossman suggests once the number of functions that a program has is known, then an estimate of man-hours required for development can be determined.

An advantage to the model is its simplicity. Project variables do not have to be identified and their influence estimated. This removes part of the subjectivity that is incorporated within many other models. Conversely, a disadvantage is that you must know or be able to estimate the number of functions within a program.

C. USER FUNCTIONS

A third area of measurement uses the number of inputs, inquiries, outputs, and master files delivered to the user to determine programmer productivity [Ref. 23]. Each category by program is counted, weighed, aggregated, and adjusted for complexity. The delivered results is a dimensionless number in function points, which when compared to a data base of

like measures provides an indication of the relative user value.

Albrecht's metric looks particularly attractive because for the first time a model attempts to measure output, namely, user functions. As a result, quality measurement is less of a problem than with other metrics because user interaction is incorporated within the metric. For the same reason, the model is less susceptible to programmer gaming. Another advantage of this model is its apparent language portability. One possible problem with the metric is the subjective determination of whether a function is an input or an inquiry. This decision may critically influence the model if different weighting factors are used for the two categories. The literature contains no information about the model's ability to measure productivity (other than what Albrecht provides); however, because of the advantages the model offers it warrants strong consideration for testing.

D. MODEL RECOMMENDATIONS

As the first step in choosing a metric for measuring programmer productivity at the Fleet Material Suppport Office (FMSO), it is recommended that three of the discussed metrics be tested for performance using FMSO data. The recommended metrics are: (1) Boehm's COCOMO model (basic), Johnson's averages for different length programs (lines of code per hour), and Albrecht's user function model. These models were

selected for three primary reasons. First, their relative simplicity of design and ease in testing (Johnson's and Boehm's model) make them attractive for further evaluation. Secondly, they provide a good cross-section of not only the production function but also of the available published models. Finally, it is believed the delivered results from one or a combination of these three models may suffice FMSO's various measuring and predicting requirements. Accordingly, it is recommended the models also be tested under various environments (ie., new application software development, maintenance, etc.) in order to determine their accuracy and usefulness as an indicator of programmer productivity for specific FMSO applications. As possible, the results from each of the models will be evaluated for predictability of measurement and ease of implementation at FMSO.

IV. METRIC TEST

Boehm's, Johnson's and Albrecht's productivity measuring metrics, which were recommended for testing in Chapter Three, are evaluated in this chapter using FMSO data. Two separate productivity measurement experiments were conducted: (1) on new application software development and (2) on the maintenance of existing programs. The first experiment was conducted on a project consisting of fourteen programs (Requistion Response Management Information System [RRMIS]). Data elements required for each of the three models were collected on this database and evaluated. In the second experiment, a database consisting of thirty programs was used. As before, the programs constitute a larger project (MISIL). In the second experiment only Boehm's and Johnson's models were to be tested. Albrecht's model does not lend itself to measuring productivity in the maintenance environment and, therefore, was not included. The intent in both experiments was to evaluate the predictive ability of the above mentioned models using represenative FMSO data and to determine if further research appears warranted.

A. DEFINITIONS

In the experiment on new application software development, data elements were collected on: (1) lines of

code, (2) time actually spent in development and (3) on function points delivered or designed (as defined by Albrecht). Lines of code or delivered source instructions (DSI) was defined to "...include all program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers, and assemblers. It excludes comment cards and unmodified software. It includes job control language, format statements, and data declarations. Instructions are defined as lines of code or card images...." [Ref. 24]. This description/definition of a line of code was used consistently throughout all the experiments.

The second data element, time spent in the development process, is the time actually spent in man-hours in the design and implementation of the software programs. In other words, it is the time spent between the beginning of the product design phase and the end of the implementation/ integration phase. This is not an aggregate measurement in that it does not include overhead costs (ie., vacations, sick time, non-related meeting time, etc.). Throughout the experiments, the definitions of man-days and man-months that are presented in the COCOMO model are used. They are as follows:

MAN-DAY (MD).....8 HOURS OF WORK

MAN-MONTH (MM).....152 HOURS OF WORK
(OR 19 MD)

The third data element, delivered function points, is defined in accordance with the guidance provided by Albrecht [Ref. 25]. An example of the definition of terms and the worksheet used in their calculation can be found attached as Appendix A.

B. TEST PROCEDURES (APPLICATION SOFTWARE)

Using the data elements from the common database (attached as Appendix B), all three models were exercised in accordance with respective instructions. The first metric tested was the COCOMO model. To employ this model, it must first be calibrated with the user's programs. This is necessary because the model's results are dependent upon the database used in deriving the effort formula (Formula 4-1). Accordingly, if the model is not calibrated, the results will not accurately project specific program effort, and thus specific user productivity. To calibrate the model for this experiment, actual development time in man-months and program length in KDSI were converted to natural logarithms for half of the sample database. This was required in order to linearize the data for statistical analysis. A regression was then performed between ln(KDSI), the independent variable (X), and ln(MM), the dependent variable (Y). The resulting regression line was used in modifying the given COCOMO effort

$$\text{EFFORT:} \quad MM = 6.18(KDSI)^{.088} \qquad \text{FORMULA 4-1}$$

equation to reflect the programs being measured. The steps used in the calibration are provided in Appendix C. As required in Boehm's basic model, the delivered source instructions (KDSI) to be tested were then used to estimate the number of man-months (MM) required for the software development phase of the life cycle (Formula 4-1). The second calculation conducted using the COCOMO model is a productivity estimate. Productivity is defined as deliverable source instructions divided by effort (as received from Formula 4-1). Formula 4-2 shows the calculation involved.

$$\text{PRODUCTIVITY: } \frac{\text{DSI}}{\text{MM}} = \frac{\text{DSI OF PROGRAM}}{\text{EFFORT}} \qquad \text{FORMULA 4-2}$$

It should be noted that this is a partial factor measurement of an intermediate product, and as such has the deficiencies stated in Chapters Two and Three. The received results from the sample database for this model are attached as Appendix C. A comparison between actual and derived productivity results was made.

The procedures used on Johnson's model are straightforward. The total time spent in developing and implementing each program was divided into the total source instructions delivered. This is also a partial factor and an intermediate measure of productivity. Appendix D lists the

obtained results in two formats: (1) lines of code delivered per man-day and (2) lines of code delivered per man-hour (one man-day equals eight man-hours).

The procedures used in Albrecht's model follow the guidelines provided on his worksheet (Appendix A). In each of four categories (inputs, outputs, files and inquiries) function points that are delivered by or designed into the program were counted. The individual totals were then weighted and summed (unadjusted function points). Next, a modifying complexity adjustment was determined. This value is derived by making subjective determinations in ten complexity categories (0-5 scale, with 0 equalling none and 5 equalling essential). The product of the two calculations is a function point value that the program returns to the user. Caution must be exercised in using this model for when this value is plotted/compared against development time, it may wrongfully be construed as a rough indication of productivity. In fact, the model is designed to be a relative measure against an existing database. Results from Albrecht's model should be viewed as a measure of value given to the user. As discussed in previous chapters, the model attempts, for the first time, to measure the final output of the software development process. Appendix E provides the obtained results.

C. TEST RESULTS (APPLICATION SOFTWARE)

The first model tested was Barry Boehm's COCOMO model. Using the calibrated/given formulas, productivity (DSI/MM) was calculated for the last seven programs in the database (the first seven were used to calibrate the model). For the programs tested, the COCOMO model was found to be a fair estimator of productivity. In the best case a productivity of 89 DSI/MM was estimated and 91 DSI/MM was actually achieved. In the worst case, 29 DSI/MM was estimated and 70 DSI/MM was actually attained. When actual productivity (X) and estimated productivity (Y) were used in a regression, the plot in Figure 4.1 resulted. As can be seen, the data points grouped nicely around the regression line. The correlation coefficient between the values was .96, indicating a strong linear relationship exists (cause and effect relationships are not implied and cannot be assumed from these results). As can be seen in Appendix C, the actual and estimated productivity values were not clustered around any one point. The estimated productivity values ranged from a high of 590 to a low of 29 DSI/MM (mean equals 207.4, sample standard deviation equals 180.4). The actual productivity ranged from a high of 536 to a low of 70 DSI/MM (mean equals 199.4, sample standard deviation equals 158.0).

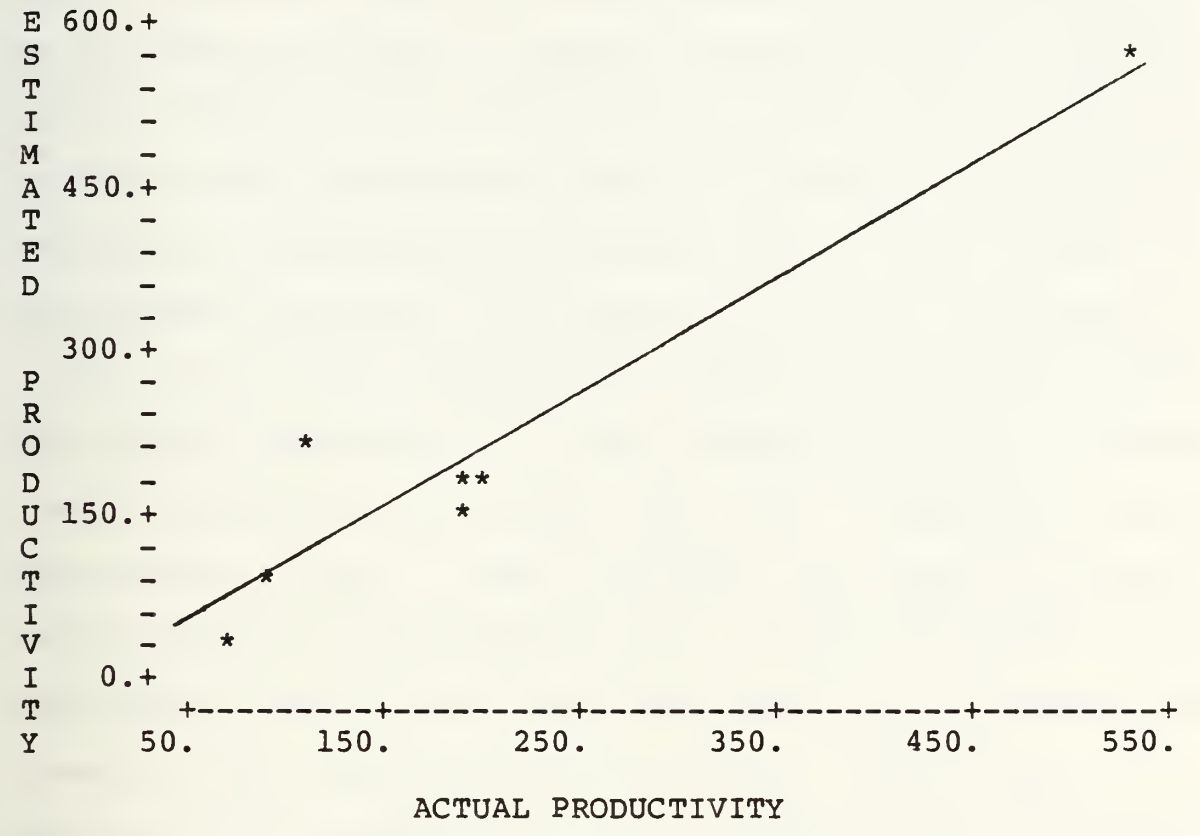

Figure 4.1 Estimated to Actual Productivity Regression (Boehm's Model)

draw conclusive statistical evidence. Still, there is encouragement that the COCOMO model can estimate programmer productivity at FMSO. Additionally, it is anticipated that with the use of a more advanced version of the COCOMO model the results could be better (there are three levels of the COCOMO model, the most elementary of which was tested).

The second model tested was Johnson's lines of code model. Program lengths were divided by the time spent in

their development. The results were lines of code per man-day or man-hour. Once the calculations were completed for the fourteen programs, a linear regression was done between lines of code (LOC) per man-hour and program size in order to determine the closeness of the relationship. Program size was used as the independent variable and LOC per man-hour as the dependent variable. The results are shown as Figure 4.2.

The correlation coefficient between LOC/MH and program size was an impressive .97, which suggests there is a strong linear relationship between the two. Supporting this observation is the data in Figure 4.2, which is nicely grouped around the regression line. For the data used in the experiment, program size would have been a fair predictor of lines of code produced per man-hour. This should not, however, be interpreted to mean program size is a good indicator of programmer productivity. There are many reasons why this may not be true. For example, gaming may occur or the programming language may be different.

Although the results should not be used to measure programmer productivity, it may be useful as an indicator of problem areas. Programs whose line of code per man-hour are substantially different from the mean should be investigated to determine the causes (ie., program complexity, programmer inefficiencies, etc.). On balance, whether this relationship holds up on a broader scale is unknown and probably should be

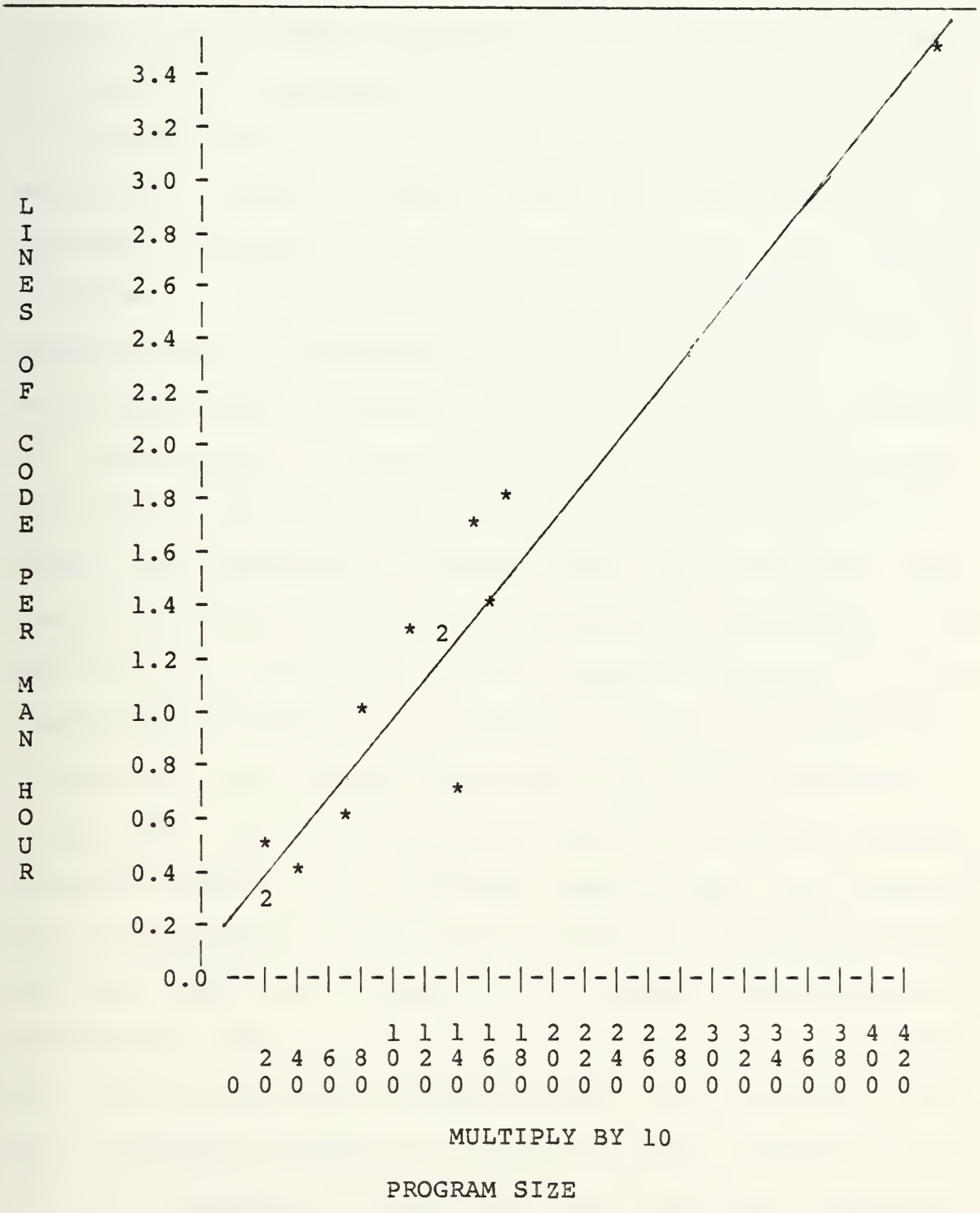

Figure 4.2 Lines of Code to Program Size Regression

tested. The confidence intervals on the estimated productivity (LOC/MH) values which the regression provided are attached as Appendix F.

The third metric tested was Albrecht's model. Function points (ie., inputs, outputs, files and inquiries) for the fourteen programs were calculated using the model's worksheet. A linear regression was then performed between program length (independent variable) and delivered function points (dependent variable). Figure 4.3 shows the results of the regression. Initially, the two variables had a correlation coefficient of .07, which indicated there was almost no relationship between the two. However, upon inspection it was noted that data point 11 was unique. Not only is it an "outlier" on the regression plot, it also stands out as different in Johnson's model. In the latter, it was the only program above the value of 2 LOC/MH (its value was 3.6; mean equals 1.16 and sample standard deviation equals .87). Although investigated, the reason(s) for its uniqueness could not be determined. Appropriately, the data point was removed and a second regression was performed. The results are shown in Figure 4.4. As seen, the data is obviously grouped around the regression line. The correlation coefficient between the two variables is .63. This is a substantial improvement from the first regression. Still, a .63 correlation value indicates only a weak

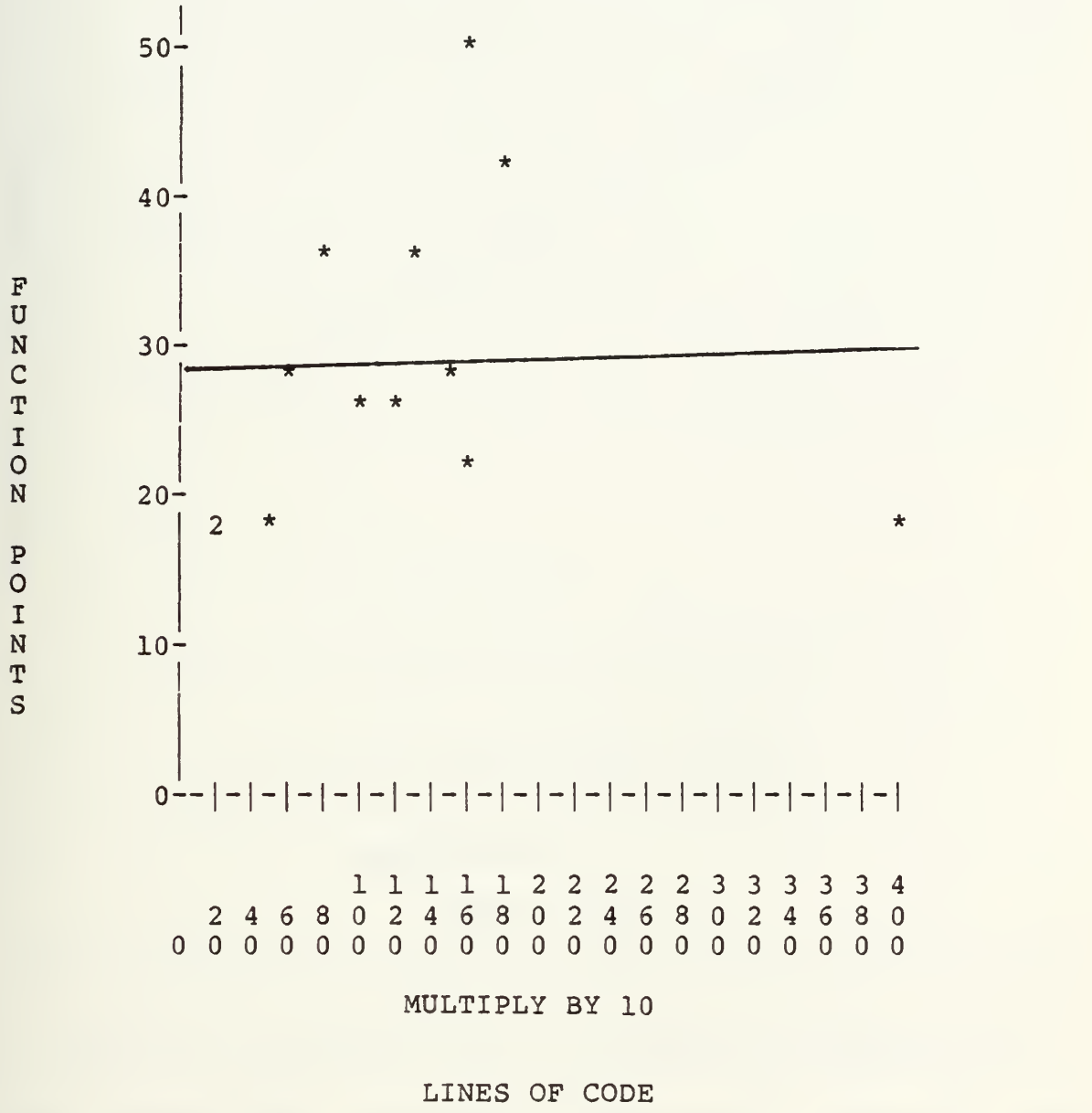

Figure 4.3 Function Points to Program Size Regression

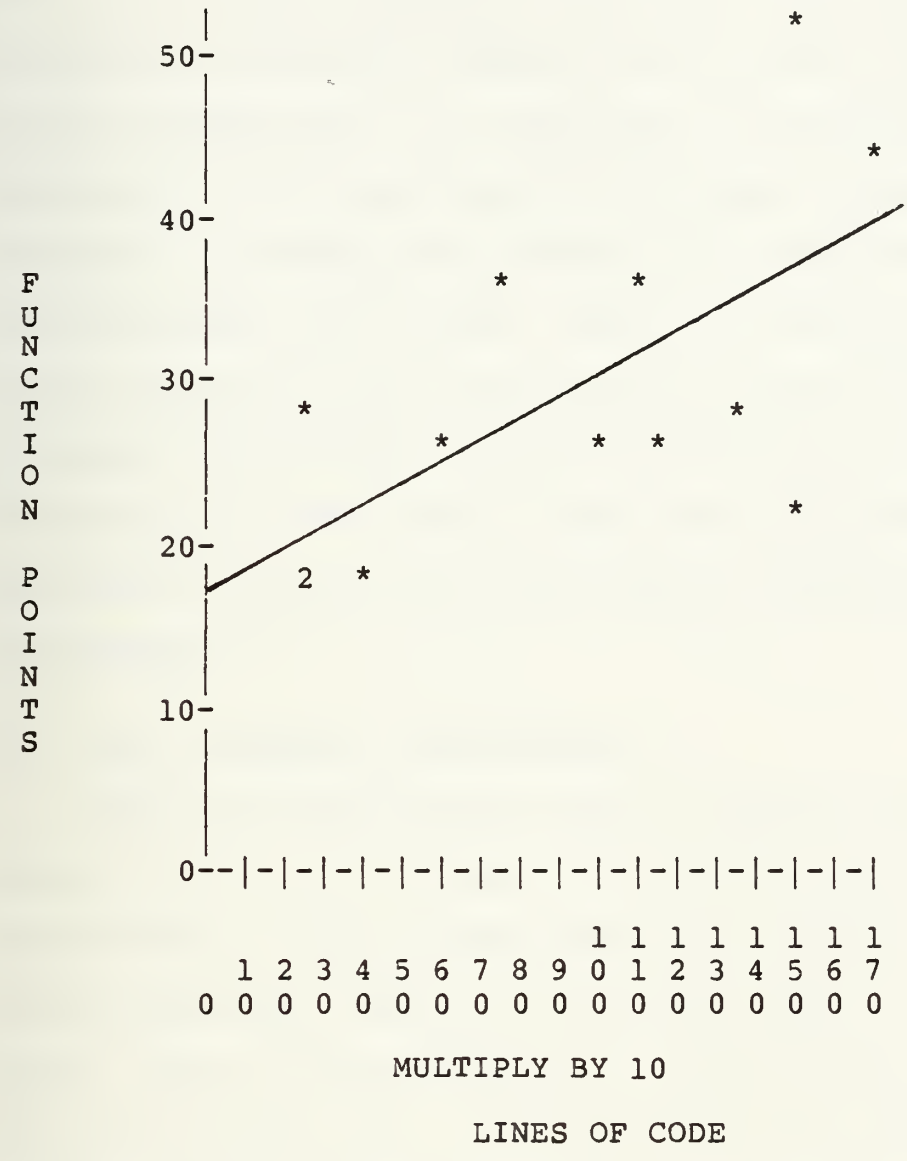

Figure 4.4 Function Points to Program Size Regression

relationship (.85 and above is desirable). Possibly, the size of the database and the length and type of the programs used affected the results of this test. It should be noted that although the results of the regression are interesting

and suggestive, Albrecht's model is designed to be a relative and not a linear measure, ie., only when compared with a database of historical function point counts for similar type projects/programs is a particular productivity measure meaningful. It must be put into proper perspective. This could not be accomplished during this experiment because no such database now exists. There is, however, sufficient encouragement from the results of this experiment to recommend that further tests of this model be conducted on a broader scale with similar data and evaluated. The derived confidence intervals from the second regression are attached as Appendix G.

D. TEST PROCEDURES (MAINTENANCE)

The second experiment tried to measure programmer productivity in the software maintenance (and enhancement) environment. The database used for the test consisted of thirty programs ranging in size from 496 to 10,203 lines of code. The maintenance activity measured were all changes made to existing code. The modifications ranged from a low of 13 to a high of 915 changed lines. The degree of change was between one and sixty-five percent of program length. Appendix H lists the maintenance data used for the experiments.

Two metrics were to be evaluated, Boehm's COCOMO and Johnson's lines-of-code models. Unfortunately, Boehm's model

could not be tested because required data was unavailable. In order to use Boehm's metric, in the maintenance environment, it is necessary to first calibrate a basic effort equation (similar to Formula 4-1) using the original program's actual development times (MM) and lengths (KDSI). While for the MISIL data program lengths were known, their original development times were not. Therefore, the basic effort equation could not be derived and a meaningful test of the model's predictive abilities could not be performed.

Once the basic effort equation has been derived for Boehm's model, the annual change traffic (ACT) value must then be calculated. Formula 4-3 applies.

$$ACT = \frac{\text{DSI ADDED + DSI MODIFIED}}{\text{TOTAL OLD DSI}} \qquad \text{FORMULA 4-3}$$

The ACT figure is "...the fraction of software product's source instructions which undergo change during a (typical) year, either through addition or modification...." [Ref. 26]. The COCOMO model multiplies the ACT value times the applicable estimated development effort value received from the basic effort equation in order to determine the estimated annual maintenance effort (Formula 4-4). The COCMO derived annual maintenance effort should then be compared against the known annual maintenance time in order to see how well it can predict.

FORMULA 4-4

$$MM_{AM} = 1.0\ (ACT)\ (MM_{D})$$

MM_D .....ESTIMATED DEVELOPMENT EFFORT

MM_{AM} .....BASIC ANNUAL MAINTENANCE EFFORT

The second metric to be tested in the maintenance environment was Johnson's lines-of-code per man-hour measurement. For each program the changed lines of code were divided by the man-hours expended in making the change. Useful patterns/trends were then looked for which might help management in decision making.

E. TEST RESULTS (MAINTENANCE)

As discussed, the only model tested in the maintenance environment was Johnson's lines of code metric. Lines of changed code were divided by the man-hours spent in making the changes. The results, lines-of-code per man-hour, were then scanned for predictability/useability. The resulting data is shown in Appendix I. As can be seen, lines-of-code per man-hour ranged from a high of 6.6 to a low of .2. The mean was 1.7, with a sample standard deviation of 1.6. The correlation coefficient between change size (LOC) and lines-of-code per man-hour was .69. A further review of the data did not reveal any patterns or trends which might be useful

to management. In fact, the derived data appeared to be near random in nature (a .69 correlation is not strong enough to be useful). Accordingly, it is recommended this model not be strongly considered for further evaluation.

F. MANAGEMENT CONSIDERATIONS

It appears, based on the discussed model results, that there is more hope in measuring programmer productivity in application development than in the maintenance environment. Johnson's model, the only model actually tested in the maintenance environment, did not return meaningful or useful data. Boehm's model may be better and should probably be tested in further evaluations. Still, Boehm's model relies on the derived effort equation and the annual change traffic (ACT) in order to determine the estimated annual maintenance effort. Any error in the basic effort equation will be compounded by later calculations and reflected in the final result. There is simply more room for error in Boehm's maintenance than in his application productivity measuring metric. In comparison, all the models tested on application development software (Boehm's, Johnson's and Albrecht's) showed promise. Boehm's model did a fair job of estimating programmer productivity. However, as previously stated, the tested database was too small to be conclusive. Still, there is an indication that the model can be useful. Just how useful and in what areas (planning, control or evaluation)

will depend on the results of further testing. In all these areas management should be careful not to draw unsupported conclusions from the results of this model. It is imperative that Boehm's model be carefully tested and proven reliable before it is used. Johnson's model provided a good linear relationship between lines-of-code per man-hour and program length. The knowledge of such a relationship can be useful to management in two ways. First, it can help to identify program areas ahead of time that take longer to develop. With such knowledge, management can plan accordingly. Seondly, programs already in the development process that require managerial attention can be identified sooner (ie., programs that take longer/shorter than normal time to develop). This knowledge allows management to reprogram effort in a more timely manner. The third model that showed promise in the application environment was Albrecht's metric. Although further testing and evaluation is required before a determination can be made as to it's specific usefulness, there is encouragement from the experiment's results. On balance, the strongest factor supporting further experimentation with this model is the still unsatisfied need to accurately measure programmer productivity. This model because it measures an output and not an intermediate product still appears to offer the best hope of satisfying that requirement.

Accordingly, it is recommended that FMSO consider collecting in a routine manner the data elements required for all three application environment models and for Boehm's metric in the maintenance environment so that further testing and evaluation can be accomplished. Also, it is recommended that additional tests try to identify practical FMSO applications for the derived productivity data and measure/quantify received benefits. This type of information must also be known if a rational decision based upon cost and benefits is to be made concerning the implementation of a productivity measure at FMSO.

V. PRODUCTIVITY PERSPECTIVES

Once a programmer productivity metric has been selected, calibrated, tested and proven reliable, management may ask what specific variables affect productivity and to what degree can they be influenced. They may also ask if it is possible to precisely predict the results of planned change. For example, will four programmers assigned to a project produce twice as much as two (or cut the development time in half), or will productivity increases justify the cost of new software productivity tools (ie., is the return on the investment sufficiently large). These are not trival questions and answers are not easily derived. However, they are critical questions because they determine proper areas where managerial attention must be focused and corporate capital should be invested. Additionally, to an extent, they drive organizational goals and objectives. As might be expected, judgement errors in this area are often embarrassing, costly and dangerous. Because of the severity of the impact, before change is implemented influencing variables must be carefully examined and analyzed to ensure the desired result is achieved, and that ripple effects are not counterproductive. Where the desired result cannot accurately be estimated, which is normally the case, management must be aware of the risk involved.

The variables within the programmer environment, which management can influence, can be classified into four organizational categories: (1) management, (2) environment, (3) people and (4) the process [Ref. 27]. Each of these very aggregate areas and how they relate to programmer productivity will be presented in this chapter. Additionally, within each category specific elements and their impact, which are discussed in the literature as increasing programmer productivity, will be included.

A. MANAGEMENT

Management must set the stage for achieving gains in programmer productivity. They must create a climate with open communication lines that is conducive to change. This can only be accomplished if the managers (at all levels) have the appropriate knowledge of technical and administrative requirements and are able to prioritize the urgency of the various undertakings. Improving programmer efficiency is one of those requirements, and unless management strongly and actively emphasises its importance gains in productivity will not be realized.

Management must not only assign a high priority to improvements in productivity, they must also make sure that appropriate awards and incentives are in place. Even more importantly, they must make sure the rewards that are in place are not counterproductive. An example of the latter

may be rewards based upon the number of completed up and running programs or lines of code written by a programmer. These rewards can be dysfunctional in that they encourage quantity with no measure of quality. Management must ensure rewards encourage real improvement.

Resources are always scarce. Management's role in software development is to optimally use those scarce resources in the production of code. This requires not only properly rewarding people for superior effort but also using their individual talents and expertise in the most economically efficient manner possible. One managerial organizational approach that has enjoyed some success (mixed reviews) is the use of a chief programming team. Under this concept, a senior programmer with proven performance is responsible for the detailed development of the programming effort. He is supported by additional programming personnel with lesser skill, and often an assistant chief programmer, a program librarian, and clerical assistance.

The concept recognizes two important qualitites about programmers specifically and people in general. First, that there are different levels of competence and expertise among programmers. Barry Boehm in his article, "Seven Basic Principles of Software Engineering", made the point that the chief programmer may be five or more times more productive than the lowest member of the team [Ref. 28]. Accordingly,

to achieve maximum technical and economical efficiency, the most competent programmers should be assigned the major or most complex part of the work. Other programmers should serve in supportive roles. This approach dovetails nicely with the desired awards structure. Outstanding and improved performance can be recognized and rewarded.

The second recognized concept is span of control. As might be expected, the chief programmer's area of responsibility in this structure is clearly defined and, therefore, can be of manageable size. Experience indicates that ten people should be the upper bound for the programming team [Ref. 29]. As a result, communication and coordination problems that are so often associated with software development can be minimized resulting in direct cost savings. Additionally, this structure allows management to more closely monitor the project's headway, facilitating earlier problem identification and correction. This, in turn, further increases productivity.

Although the chief programming team concept of management offers obvious advantages it also has noted deficiencies. First, it relies heavily upon the chief programmer for success. If his managerial and/or technical skills are weak then there is a good chance of failure. Conversely, if the chief programmer's skills are particularly strong then there is a good chance he will be offered other jobs and will not

complete the project. The demand for individuals with these talents is strong. The assistant chief programmer can partially make up the difference in both scenarios; however, he too can be weak/lost. An additional problem is incompetency. In a small team environment each player is critical. The loss of even one member seriously affects the chances for success. Management must decide if the organizational infrastructure and the nature of the work make this method of management a viable and attractive alternative.

As problems arise and decisions are made in the software development process, management must be aware of the inherent pitfalls. For example, a continuing managerial problem is the schedule. As problems develop and programs fall further and further behind, management's natural tendency is to add more and more programmers in order to get well. This can create an emotional tail-chasing situation. Dr. Fred Brooks pointed out the paradox of the problem in his article, "The Mythical Man-Month" [Ref. 30]. By adding manpower to a project that is already late a counterproductive situation can occur. New people thrown into the middle of a project about which they know nothing require assistance from the experienced to get started. This assistance comes at the expense of still further slippage in the schedule. If management tries to compensate for the additional slippage by

adding still more people a vicious never-ending descending spiral to failure can develop.

B. ENVIRONMENT

Capable and motivated employees can only perform to the limits of their abilities (technical efficiency) if they have the necessary tools and proper environment in which to work. A programmer that has adequate desk space, the required tools and a relatively quiet area will be much more productive than his counterpart who works in a noisy congested office with inadequate tools. The environment is a ripe area for productivity capital investments in most companies because the marginal return is likey to be large. There are many areas where management can make productive environmental improvements. For instance, they can ensure there are adequate phone and computer terminals available. A substantial amount of productive time can be lost if the programmers must constantly wait in line for these services. Other improvements in programming efficiency can be made through the use of sign-out boards and by supplying adequate clerical and administrative support. The environment is extremely important to productivity and must not be overlooked.

C. PEOPLE

Considering the current and ever-expanding shortage of programmers and their upward spiraling wages, people problems

may be management's major concern. Not only is there a shortage of available programmers, there is also a vast range of differences in their abilities. Figure 5.1 shows the results of a small study (based on twelve programmers) done by H. Sackman, W. J. Erickson and B. G. Grant on programming performance using a time sharing on-line programming approach compared to the more classical batch style of programming. It should be noted that the on-line process was accomplished

PERFOMANCE MEASURE	POOREST SCORE	BEST SCORE	RATIO
1. DEBUG HOURS ALGEBRA	170	6	28:1
2. DEBUG HOURS MAZE	26	1	26:1
3. CPU TIME ALGEBRA (SEC)	3075	370	8:1
4. CPU TIME MAZE (SEC)	541	50	11:1
5. CODE HOURS ALGEBRA	111	7	16:1
6. CODE HOURS MAZE	50	2	25:1
7. PROGRAM SIZE ALGEBRA	6137	1050	6:1
8. PROGRAM SIZE MAZE	3287	651	5:1
9. RUN TIME ALGEBRA (SEC)	7.9	1.6	5:1
10.RUN TIME MAZE (SEC)	8.0	.6	13:1

Source: Parikh, G., How to Measure Programmer Productivity, p. 35, Shetal Enterprises, 1981.

Figure 5.1 Range of Individual Differences in Programming Performance

may be management's major concern. Not only is there a shortage of available programmers, there is also a vast range of differences in their abilities. Figure 5.1 shows the results of a small study (based on twelve programmers) done by H. Sackman, W. J. Erickson and B. G. Grant on programming performance using a time sharing on-line programming approach compared to the more classical batch style of programming. It should be noted that the on-line process was accomplished

PERFOMANCE MEASURE	POOREST SCORE	BEST SCORE	RATIO
1. DEBUG HOURS ALGEBRA	170	6	28:1
2. DEBUG HOURS MAZE	26	1	26:1
3. CPU TIME ALGEBRA (SEC)	3075	370	8:1
4. CPU TIME MAZE (SEC)	541	50	11:1
5. CODE HOURS ALGEBRA	111	7	16:1
6. CODE HOURS MAZE	50	2	25:1
7. PROGRAM SIZE ALGEBRA	6137	1050	6:1
8. PROGRAM SIZE MAZE	3287	651	5:1
9. RUN TIME ALGEBRA (SEC)	7.9	1.6	5:1
10.RUN TIME MAZE (SEC)	8.0	.6	13:1

Source: Parikh, G., How to Measure Programmer Productivity, p. 35, Shetal Enterprises, 1981.

Figure 5.1 Range of Individual Differences in Programming Performance

more quickly but at the expense of cpu cycles. Management must constantly conduct cost benefit analysis on these types of tradeoffs in order to determine optimum efficiency (classical capital labor tradeoff). Because of this apparent vast difference in performance, it is essential that management develop skill profiles for each classification area, ie., analyst, programmers, etc.. Accordingly, both management and the individual employees should on a continuing basis assess themselves against these requirements. Where deficiencies are noted, training programs should be encouraged/offered. Management in today's environment must groom their people to be more productive and encourage upward mobility [Ref. 31].

Programmers, like other people, need to have goals and objectives to work towards. Management must not only prioritize programming requirements, they must also establish achievable and measurable goals for productivity improvement. The importance of this requirement was highlighted in an experiment conducted by Gerald H. Weinberg in 1971-2. The experiment tried to assess the effect clear goals have on performance. Figure 5.2 shows the experiment's results. As can be seen, when management made clear the programming objectives they were attained (a scale of 1 to 5 is used in which 1 is optimum and 5 is less than optimum goal achievement). It should be noted from the results of this

experiment that there can be conflicting goals. For example, core minimization and output clarity appear to be diametrically opposing goals. In these cases, management must be aware of the problem, decide the tradeoff and state the organizational policy. Programmers can meet objectives only if they know what is expected of them. According to Weinberg, studies such as this dispel the myth that there are "good and horrid" programmers. Based upon this and other related experiments the following major conclusions were drawn by Weinberg from their endeavors [Ref. 32].

RANKING

GROUP OBJECTIVES	CORE	OUTPUT CLARITY	PROGRAM CLARITY	STATEMENTS	HOURS
MINIMUM CORE	1	4	4	2	5
OUTPUT CLARITY	5	1	1-2	5	2-3
PROGRAM CLARITY	3	2	1-2	3	4
MINIMUM STATEMENTS	2	5	3	1	2-3
MINIMUM HOURS	4	3	5	4	1

Source: Parikh, G., How to Measure Programmer Productivity, p. 36, Shetal Enterprises, 1981.

Figure 5.2 Ranking of Programming Performance on Five Objectives

1. Programming is such a complex activity that programmers have an almost infinite number of choices in terms of how they will write a program in order to meet certain objectives.

2. If given specific objectives, programmers can make programming choices in such a way that they will meet those objectives--provided they do not conflict with other specific objectives.

3. Programmers adjust their estimates, depending on what goals are stressed, to give themselves more "cushion" for meeting stressed goals.

4. Time to complete a program need not be critical if adequate time is allowed, but in no case should experimental results be mixed if some programmers felt pressed for time.

5. Optimization goals tend to be highly confliciting with other goals, even with the primary goal of correctness.

6. No programming project should be undertaken without clear, explicit, and reasonable goals.

7. No experiment on programmer performance should be undertaken without clear, explicit, and reasonable goals--unless that experiment is designed to measure the effect of unclear, implicit, or unreasonable goals.

D. PROCESS

In the actual writing of software code there are two ways producitvity can be increased: (1) through a change in the activities of the programmer and (2) with the addition of new equipment or tools. An example of the former is the development over the last several years of structured programming techniques. Within the structured programming concept are three generally accepted subsets: (1) structured programming coding techniques, (2) top-down program design and (3) chief

programmer teams [Ref. 33]. These methodologies of writing and constructing code evolved as a result of general weakness in previous approaches to software systems managment and development. Whether or not these principles are used by an organization in the production of code depends upon what the codes intended usage will be. If a small program is to be constructed to run one time locally, then the extra cost involved in writing the more structured code is probably not justified. However, if the program will be exported to other organizations, have a long life or contain parts that have universal application then structured programming techniques should be utilized.

There are several productivity related reasons why structure programming should be required by management. First is the problem of program maintenance and enhancement. Programs written using structured programming are much easier to understand than straight line code because the flow of logic is clearer. This is so because the interfaces between the modules is minimized and explicitly stated (loose coupling). Additionally, like procedures are grouped together to form highly cohesive modules. These techniques along with the principles of information hiding allow programs to be modified much easier than in the past. This is extremely important in view of the fact that the cost of software

maintenance is commonly the most expensive phase in the program life cycle [Ref. 34].

A second reason for using structured programming techniques is that it allows for easier reuse of code. Using structured programming techniques Raytheon was able to reuse existing code between 40 and 60 percent (average) of the time in the construction of over 500 programs. Additionally, they were able to increase the maintainability of three thousand old programs. Obviously, not all organizations can achieve such results; still, there are substantial productivity gains that can be realized in most organizations by making an effort to reuse code whenever possible [Ref. 35].

The most often looked to solution for increasing programmer productivity are aids and tools: test generators, reconcilers, disk space managers, utility tools, etc.. When management considers the acquisition of these devices, the questions naturally asked are how much will this device increase productivity, will the increase be enough to justify it's cost and how can I be sure that the the benefit is received. These questions cannot be easily answered. If a thoroughly tested and calibrated metric is in use, such as Barry Boehm's COCOMO model, then it may be possible to get a rough estimate of a tool's impact on productivity by looking at the effort multipliers influence. Anything beyond this rough estimate is risky speculation. If a metric is not up

and running, then an educated estimate is probably the best that can be achieved.

T. Capers Jones in his article, "The Limits of Programming Productivity" [Ref. 36], discused the various ways of achieving programmer productivity gains and roughly categorized how much gain could be achieved from various implementations. The groupings used were methods that may yield: (1) 5-25 %, (2) 25-50 %, (3) 50-75 % and (4) over 75 % improvement. Obviously, these groupings are extremely rough; however, they may still be useful in determining the types of things which must be done in order to achieve desired levels of programmer productivity improvement.

Prototyping is a new concept that is being looked at to increase productivity in the software development process. Unlike the step-by-step structured approach that is commonly used (feasibility, requirements, design, code, integration and implementation), prototyping puts a small subset that captures the essential features of the required program into the hands of the user immediately. The user works with this program and provides feedback to the software designers concerning desired improvements and enhancements. These changes are incorporated and the program is returned for further evaluation. This iterative process continues until allocated resources are expended or the user is satisfied. This approach to software development may offer several

advantages as compared to traditional methods. First, it gets something in the hands of the user right away. Under the structured development process it may take years before a program is provided to the user. Secondly, it requires the user to get intricately involved. This is extremely important, for as the user gets more involved his requirements become better defined. This results in the development of a software program that better meets the required needs. Accordingly, since the maintenance phase is the most expensive part of the software life cycle, any reduction of maintenance/enhancement activity will increase overall productivity and substantially lower life cycle costs [Ref. 37].

E. IMPROVEMENT PROJECTIONS

In the management of computers and programmers there are very few certainties. How much productivity will be gained by making specific changes is often unknown because the process is too complex and the results are too hard to measure precisely. Discrete measurements of programmer productivity are almost impossible to accomplish. The best management can hope to do is make educated estimates. How good the estimates are depends on management's experience, knowledge and expertise in software development. In order to develop these managerial skills, management must continue to try and measure aggregate programmer productivity. Only by

continued measurement and comparison effort can performance be judged and insight be gained into this complex issue.

VI. CONCLUSIONS AND RECOMMENDATIONS

This paper has explored the ability of various metrics to predict programmer productivity at the Fleet Material Support Office (FMSO). It has shown, through the use of the production function, that most productivity measurement metrics have severe deficiencies due to their intermediate and partial measure of productivity. If management wants to use one of these models, they must do so with the understanding and awareness of these limitations.

Based on the experiments conducted in this paper, it appears that programmer productivity measures can be useful as a managerial tool. Just how useful is unknown and will require further testing and evaluation. It is suspected, however, that one productivity metric will not meet all needs. It is likely that different models will be required to measure different areas of software development. Also, it is expected for any particular software program that different metrics will be required depending on the use of the data, ie., programmer evaluation, program planning, etc.. Programmer productivity metrics will probably demonstrate different predicting abilities between program types and application usages.

In view of the above, it is recommended that FMSO gather data elements on various program types and continue to test

several programmer productivity metrics. If possible, the selected metrics should try to measure final program output. The results from these tests should be evaluated in two areas: (1) on how well the metric predicts actual productivi ty and (2) on how useful the derived data is for the intended application. Additionally, it is recommended that FMSO identify specifically the benefits to be received from measuring productivity and determine the cost it is willing to pay. With that decided, rational decisions can better be made as to the model(s) selection.

IBM DP SERVICES — Date: _____

FUNCTION VALUE INDEX WORKSHEET — Project ID: _____

Project Name: _____

Prepared by: _____ Date: _____. Reviewed by: _____ Date: _____.

Project Summary: Start Date _____ End Date _____ Work-Hours _____ Function Points Delivered or Designed _____. (from calculation).

Function Points Calculation (Delivered or Designed):

Note: Definitions on back of form.	Allocation estimated by Project Manager: Delivered by New Code	Delivered by Modifying Existing Code	Delivered by Installing and Testing a Package	Delivered by Using a Code Generator	Totals (Identify Preponderant Language)		
Language							
Inputs						X 4	
Outputs						X 5	
Files						X 10	
Inquiries						X 4	
Work-hours						Total	
Design						Unadjusted	
Implementation						Function Points	

Complexity Adjustment: (Estimate degree of influence for each factor)

___ Reliable backup, recovery, and/or system availability are provided by the application design or implementation. The functions may be provided by specifically designed application code or by use of functions provided by standard software. For example, the standard IMS backup and recovery functions.

___ Data communications are provided in the application.

___ Distributed processing functions are provided in the application.

___ Performance must be considered in the design or implementation.

___ In addition to considering performance there is the added complexity of a heavily utilized operational configuration. The customer wants to run the application on existing or committed hardware that, as a consequence, will be heavily utilized.

___ On-line data entry is provided in the application.

___ On-line data entry is provided in the application and in addition the data entry is conversational requiring that an input transaction be built up over multiple operations.

___ Master files are updated on-line.

___ Inputs, outputs, files, or inquiries are complex in this application.

___ Internal processing is complex in this application.

Degree of Influence on Function:

0	None	3	Average
1	Incidental	4	Significant
2	Moderate	5	Essential

___ Total Degree of Influence (N)

___ Complexity adjustment equals (0.75 + 0.01 (N))

Unadjusted Total X Complexity Adjustment = Function Points Delivered or Designed

_____ X _____ = _____

Definitions:

General Instruction:

Count all inputs, outputs, master files, inquiries, and functions that are made available to the customer through the project's design, programming, or testing efforts. For example, count the functions provided by an IUP, FDP, or Program product if the package was modified, integrated, tested, and thus provided to the customer through the project's efforts.

Work-hours:

The work-hours recorded should be the IBM and customer hours spent on the DP Services standard tasks applicable to the project phase. The customer hours should be adjusted to IBM equivalent hours considering experience, training, and work effectiveness.

Input Count:

Count each system input that provides business function communication from the users to the computer system For example:

- data forms
- terminal screens
- scanner forms or cards
- keyed transactions

Do not double count the inputs. For example, consider a manual operation that takes data from an input form, to form two input screens, using a keyboard to form each screen before the entry key is pressed. This should be counted as two (2) inputs not five (5).

Count all unique inputs. An input transaction should be counted as unique if it required different processing logic than other inputs. For example, transactions such as add, delete, or change may have exactly the same screen format but they should be counted as unique inputs if they require different processing logic.

Do not count input or output terminal screens that are needed by the system only because of the specific technical implementation of the function. For example, DMS/VS screens, that are provided only to get to the next screen and do not provide a business function for the user, should not be counted.

Do not count input and output tape and file data sets. These are included in the count of files.

Do not count inquiry transactions. These are covered in a subsequent question.

Output Count:

Count each system output that provides business function communication from the computer system to the users. For example:

- printed reports
- terminal screens
- terminal printed output
- operator messages

Count all unique external outputs. An output is considered to be unique if it has a format that differs from other external outputs and inputs, or, if it requires unique processing logic to provide or calculate the output data.

Do not include output terminal screens that provide only a simple error message or acknowledgement of the entry transaction, unless significant unique processing logic is required in addition to the editing associated with the input, which was counted.

Do not include on-line inquiry transaction outputs where the response occurs immediately. These are included in a later question.

File Count:

Count each unique machine readable logical file, or logical grouping of data from the viewpoint of the user, that is generated, used, or maintained by the system. For example:

- input card files
- tape files
- disk files

Count major user data groups within a data base. Count logical files, not physical data sets. For example, a customer file requiring a separate index file because of the access method would be counted as one logical file not two. However, an alphabetical index file to aid in establishing customer identity would be counted.

Count all machine readable interfaces to other system as files.

Inquiry Count:

Count each input/response couplet where an on-line input generates and directly causes an immediate on-line output. Data is not entered except for control purposes and therefore only transaction logs are altered.

Count each uniquely formatted or uniquely processed inquiry which results in a file search for specific information or summaries to be presented as response to that inquiry.

Do not also count inquiries as inputs or outputs.

APPENDIX B

RRMIS DATA

PROGRAM	LINES OF CODE	ACTUAL DEVELOPMENT TIME (MM)	FUNCTION POINTS
1	1,685	6.3	41.4
2	1,547	6.2	50.6
3	395	7.1	18.4
4	248	5.4	17.0
5	245	4.9	17.0
6	1,597	7.0	22.75
7	762	5.3	35.6
8	1,004	5.2	26.7
9	1,350	12.2	27.6
10	520	5.7	26.7
11	4,129	7.7	17.8
12	1,153	6.0	36.8
13	1,156	5.7	26.1
14	153	2.2	27.65

APPENDIX C

BOEHM'S MODEL

MODEL CALIBRATION

PROGRAM	ACTUAL DEVELOPMENT TIME (MM)	PROGRAM LENGTH (KDSI)	Y ln(MM)	X ln(KDSI)
1	6.3	1.685	1.84	.52
2	6.2	1.547	1.82	.44
3	7.1	.395	1.96	-.93
4	5.4	.248	1.69	-1.39
5	4.9	.245	1.59	-1.41
6	7.0	1.597	1.95	.47
7	5.3	.762	1.67	-.27

Alpha = 1.82

Beta = .088

$$EFFORT = MM = e^{1.82}(KDSI)^{.088} \quad \text{or} \quad = 6.18(KDSI)^{.088}$$

PRODUCTIVITY MEASUREMENT

PROGRAM	KDSI	ESTIMATED EFFORT	ESTIMATED PRODUCTIVITY	ACTUAL PRODUCTIVITY
8	1.004	6.18	162	193
9	1.350	6.35	213	111
10	.520	5.83	89	91
11	4.129	7.00	590	536
12	1.153	6.26	184	192
13	1.156	6.26	185	203
14	.153	5.24	29	70

ESTIMATED PRODUCTIVITY: MEAN = 207.4

SAMPLE STANDARD DEVIATION = 180.4

ACTUAL PRODUCTIVITY: MEAN = 199.4

SAMPLE STANDARD DEVIATION = 158.0

APPENDIX D

JOHNSON'S MODEL

PROGRAM	ACTUAL MD	KDSI	LINES/MD	LINES/MH
1	118.9	1.685	14.2	1.8
2	117.0	1.547	13.2	1.7
3	135.0	.395	2.9	.4
4	103.4	.248	2.4	.3
5	92.4	.245	2.7	.3
6	133.9	1.597	11.9	1.5
7	100.9	.762	7.6	1.0
8	98.5	1.004	10.2	1.3
9	232.0	1.350	5.8	.7
10	107.4	.520	4.8	.6
11	145.4	4.129	28.4	3.6
12	113.1	1.153	10.2	1.3
13	108.4	1.156	10.7	1.3
14	41.9	.153	3.7	.5

APPENDIX E

ALBRECHT'S MODEL

PROGRAM	LINES OF CODE	FUNCTION POINTS
1	1,685	41.4
2	1,547	50.6
3	395	18.4
4	248	17.0
5	245	17.0
6	1,597	22.75
7	762	35.6
8	1,004	26.7
9	1,350	27.6
10	520	26.7
11	4,129	17.8
12	1,153	36.8
13	1,156	26.1
14	153	27.65

$\overline{X}$ = 28.00
SSD = 9.99

APPENDIX F

CONFIDENCE INTERVALS FOR LINES OF CODE

X	Y OBSERVED	Y ESTIMATED	NON-SIMULTANEOUS 95.00% CONFIDENCE LIMITS	
1685	1.8000	1.6180	1.4595	1.7765
1547	1.7000	1.5034	1.3534	1.6533
395	0.40000	0.54629	0.37234	0.72024
248	0.30000	0.42416	0.23684	0.61149
245	0.30000	0.42167	0.23406	0.60928
1597	1.5000	1.5449	1.3921	1.6977
762	1.0000	0.85119	0.70291	0.99948
1004	1.3000	1.0522	0.91261	1.1919
1350	0.70000	1.3397	1.1982	1.4812
520	0.60000	0.65014	0.48633	0.81395
4129	3.6000	3.6485	3.2025	4.0945
1153	1.3000	1.1760	1.0377	1.3144
1156	1.3000	1.1785	1.0402	1.3169
153	0.50000	0.34524	0.14858	0.54190

0.23754 = STANDARD ERROR OF ESTIMATE

20.402% OF MEAN OF Y

APPENDIX G

CONFIDENCE INTERVALS FOR ALBRECHT'S MODEL

X	Y OBSERVED	Y ESTIMATED	NON-SIMULTANEOUS 95.00% CONFIDENCE LIMITS	
1685	41.400	37.566	28.876	46.257
1547	50.600	36.006	28.332	43.681
395	18.400	22.984	16.150	29.817
248	17.000	21.322	13.486	29.158
245	17.000	21.288	13.430	29.146
1597	22.750	36.571	28.537	44.605
762	35.600	27.132	22.029	32.235
1004	26.700	29.868	24.871	34.865
1350	27.600	33.779	27.393	40.165
520	26.700	24.397	18.307	30.486
1153	36.800	31.552	26.141	36.963
1156	26.100	31.586	26.163	37.009
153	27.650	20.248	11.711	28.785

8.0591 = STANDRD ERROR OF ESTIMATE

27.990% OF MEAN OF Y

APPENDIX H

MISIL MAINTENANCE DATA

PROGRAM	SOURCE CODE TOTAL	SOURCE CODE CHANGED	MAN-HOURS EXPENDED
1	4,498	46	32
2	5,316	520	148
3	5,089	56	40
4	4,744	40	100
5	4,624	300	196
6	10,203	109	312
7	4,045	24	32
8	1,654	600	174
9	731	472	72
10	3,264	820	325
11	3,994	60	72
12	5,100	250	134
13	5,200	75	250
14	6,800	250	406
15	7,373	480	226
16	2,598	240	56
17	1,629	36	72
18	1,680	180	198
19	3,065	437	345
20	952	13	16
21	1,798	211	164
22	696	59	46
23	1,254	25	16
24	1,149	32	38
25	5,482	98	173
26	2,513	19	80
27	3,627	915	204
28	496	259	64
29	1,509	50	94
30	1,014	26	136

APPENDIX I

JOHNSON'S MODEL

PROGRAM	LINES OF CODE	MAN-HOURS	LOC/MH
1	46	32	1.4
2	520	148	3.5
3	56	40	1.4
4	40	100	.4
5	300	196	1.5
6	109	312	.3
7	24	32	.8
8	600	174	3.4
9	472	72	6.6
10	820	325	2.5
11	60	72	.8
12	250	134	1.9
13	75	250	.3
14	250	406	.6
15	480	226	2.1
16	240	56	4.3
17	36	72	.5
18	180	198	.9
19	437	345	1.3
20	13	16	.8
21	211	164	1.3
22	59	46	1.3
23	25	16	1.6
24	32	38	.8
25	98	173	.6
26	19	80	.2
27	915	204	4.5
28	259	64	4.0
29	50	94	.5
30	26	136	.2

LIST OF REFERENCES

1. Office of Management and Budget, Federal Register, v 44 no 67, p. 20556, 5 April 1979.

2. Martin, J., Design and Strategy for Distributed Data Processing, p. 200, Prentice-Hall, 1981.

3. General Accounting Office, Conversion: A Costly, Disruptive Process That Must Be Considered When Buying Computers, FGMSD-80-35, 3 June 1980.

4. Kendrick, J.W., Productivity Trends in the United States, p. 7, Princeton University Press, 1961.

5. Ibid., p. 11.

6. Boger, D.C., A Productivity Measurement System, paper written at Naval Postgraduate School, Monterey, Ca. 1983.

7. Department of Defense Instruction 7000.2, The DOD Cost /Schedule Control System Criteria.

8. Ross, D.T., Goodenough, J.B., and Irvine, C.A., "Software Engineering Process, Principles, and Goals," Computer, p. 17-27, May 1979.

9. Jones, T.C., "Measuring Programming Quality and Productivity," IBM Systems Journal, v 17 no 1, p. 52, 1978.

10. Ibid., p. 53.

11. Halstead, M.H., Elements of Software Science, p. 9-71, Elviser North Holland, 1977.

12. McCabe, T.J., "Software Complexity Measurement," Proceedings, U.S. Army/IEEE Second Software Life Cycle Workshop, p. 186-190, August 1978.

13. Curtis, B., Sheppard, S.P., Borst, M.A., Milliman, P., and Love, T., "Measuring Psychological Complexity of Software Maintenance," IEEE Transactions of Software Engineers, p. 96-104, March 1979.

14. Curtis, B., Sheppard, S.P., and Milliman, P., "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Measure," Proceedings of the Fourth International Conference on Software Engineering, p. 356-360, 1979.

15. Meals, R.R., and Gustafson, D.A., "An Experiment in the Implementation of Halstead's Measures of Complexity," IEEE Software Engineering Standards Application Workshop, p. 45-50, 1981.

16. Fitzsimmons, A., and Love, T., "A Review and Evaluation of Software Science," Computing Surveys, v 10 no 1, p. 3-18, March 1978.

17. Boehm, B.W., Software Engineering Economics, p. 57-73, Prentice-Hall, 1981.

18. Walston, C.E., and Felix, C.P., "A Method of Programming Measurement and Estimation," IBM Systems Journal, v 16 no 1, p. 54-73, 1977.

19. Jefferey, D.R., and Lawerence, M.J., "Some Issues in the Measurement and Control of Programming Productivity," Information and Management, v 4, p. 169-176, September 1981.

20. Johnson, J.R., "A Working Measure of Productivity," Datamation, v 23 no 2, p. 106-112, February 1977.

21. Crossman, T.D., "Taking the Measure of Programmer Productivity," Datamation, p. 144-147, May 1979.

22. Ibid., p. 144-147.

23. Albrecht, A.J., "Measuring Application Development Productivity," Proceedings IEEE Computer Society Conference Fall 1981, p. 232-241, 1981.

24. Boehm, B.W., p. 59.

25. Albrecht, A.J., "Measuring Application Development Productivity," Proceedings IEEE Computer Society Conference Fall 1981, p. 40, 1979.

26. Boehm, B.W., p. 71.

27. Patrick, R.L., "Probing Productivity," Datamation, p. 207- 210, September 1980.

28. Zelkowitz, M.V., "Perspective on Software Engineering," Computing Surveys, v 10 no 2, p. 204, June 1978.

29. Ibid., p. 197-216.

30. Brooks, F.P., "The Mythical Man-Month," Datamation, p. 45-52, December 1974.

31. Parikh, G., How to Measure Programmer Productivity, p. 35, Shetal Enterprises, 1981.

32. Ibid., p. 28.

33. Paretta, R.L., and Clark, S.A., "Management of Software Development," Proceedings National Computer Conference 1981, v 50, p. 349-352, 1981.

34. Chapin, N., "Productivity in Software Maintenance," AFIPS Conference/National Computer Conference 1981, v 50, p. 349-352, 1981.

35. Lanergan, R.G., and Poynton, B.A., "Reusable Code- The Application Development Technique of the Future," Proceedings of the Joint SHARE/GUIDE/IBM Application Development Syposium, p. 127-136, October 1979.

36. Jones, T.C., "The Limits of Programming Productivity," Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, p. 77-82, October 1979.

37. Nauman, J.D., and Jenkins, A.M., "Prototyping: The New Paradigm for Systems Development," Management Information System Quarterly, p. 191-194, September 1982.

BIBLIOGRAPHY

Bailey, C.T., and Dingee, W.L., "A Software Study Using Halstead Metrics," Association for Computing Machinery, 1981.

Basili, V.R., and Phillips, T., "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Performance Evaluation Review, vol. 10, Spring 1981.

Basili, V.R., "Resource Models," Models and Metrics for Software Management and Engineering, 1980.

Blumenthal, M., " Beyond Measuring Lines of Code New Gauges of Programmer Productivity," Computerworld, 28 July 1980.

Bowen, J.B., "Are Current Approaches Sufficient for Measuring Software Quality ?," ACM Proceedings of the Software Quality and Assurance Workshop, 15-17 Nov 1978.

Brooks, W.D., "Software Technology Payoff: Some Statistical Evidence," The Journal of Systems and Software, 9 March 1981.

Byars, L.L., "Solutions to Productivity Problems," Journal of Systems Management, v 33, January 1982.

Cavano, J.P., and McCall, J.A., "A Framework for the Measurement of Software Quality," ACM Proceedings of the Software Quality and Assurance Workshop, 15-17 November 1978.

Chen, E.T., "Program Complexity and Programmer Productivity," IEEE Transactions of Software Engineers, v SE-4 no. 3, 1978.

Christensen, K., Fistos, G.P., and Smith C.P., "A Perspective on Software Science," IBM Systems Journal, v 20 no 4, 1981.

Chrysler, E., "Programmer Performance Standards," Journal of Systems Management, February 1978.

Cougar, J. D., and Zawacki, R. A., Motivating and Managing Computer Personnel, John Wiley and Sons, 1980.

Curtis, B., Sheppard, S.P., Borst, M.A., Milliman, p., and Love, T., "Some Distinctions Between Psychological and Computational Complexity of Software," Proceedings, U.S. Army/IEEE Second Software Life Cycle Workshop, 21-22 August 1978.

Elshoff, J.L., "A Review of Software Measurement Studies at General Motors Research Laboratories," Proceedings, U.S. Army/IEEE Second Software Life Cycle Workshop, 21-22 August 1978.

Franklin, B., "Programmer Productivity Needs Clearer Focus," Computerworld, 26 April 1982.

Gaffney, J.E., "Metrics in Software Quality Assurance," ACM Tutorial, 9-11 November 1981.

Gilb, T., Software Metrics, Winthrop, 1977.

Gold, B., Productivity, Technology, and Capital, Lexington Books, 1979.

Greenberg, L., A Practical Guide to Productivity Measurement, Bureau of National Affairs, Inc., 1973.

Hagan, J.C., "The Productivity Implications of Performance Measurement," SHARE 53, New York, N.Y., August 1979.

Halstead, M.H. and Schneider, V., "Further Validation of the Software Science Programming Effort Hypothesis," ACM 17th Annual Technological Symposium: Tools for Improving Computing in the 80's, 15 June 1978.

Halstead, M.H., "Software Science- A Progress Report, "Proceedings U.S. Army/IEEE Second Software Life Cycle Workshop, Atlanta, Ga., 21-22 August 1978.

Hamilton, K., and Block, A., "Programmer Productivity in a Structured Environment," Infosystems, April/May 1979.

Hornbruch, F.W., Raising Productivity, McGraw-Hill, 1977.

Jones, T.C., "Productivity Measurements," SHARE 51, Boston, Mass., 20-25 August 1978.

Keider, S.P., "Why Projects Fail ?", Datamation, December 1974.

Koutsoyiannis, A., Modern Micoeconomics, Macmillian Press Ltd., 1975.

Kirkley, J.L., "Programmer Productivity", Datamation, v 23 no 5, May 1977.

Leypoldt, C.C., "Computer System Heal Thyself," Department Of Defense Institute Selected Computer Articles, 1978.

Lockett, J., "Using Performance Metrics in System Design," ACM Proceedings of the Software Quality Assurance Workshop, 15-17 November 1978.

Markham, D., McCall, J., and Walters, G., "Metrics Applications Techniques," Proceedings of Trends and Application Advances in Software Technology, IEEE:NBS., 1981.

McCall, J.A., "The Utility of Software Quality Metrics in Large-Scale Software Systems Developments," Proceedings U.S. Army/IEEE Second Software Life Cycle Workshop, 21-22 August 1978.

Osborn, R.W., "Theories of Productivity Analysis," Datamation, September 1981.

Perlis, A.J., Sayward, F.G., and Shaw, M., Software Metrics: An Analysis and Evaluation, MIT Press, 1981.

Phister, M., "A Model of the Software Development Process," The Journal of Systems and Software, February 1981.

Presser, L., "Reversing the Priorities". Datamation, September 1981.

Putnam, L.H., "Measurement Data to Support Sizing Estimations, and Control of the Software Life Cycle," IEEE Computer Society Conference Proceedings, Spring 1978.

Putnam, L.H., and Fitzsimons, A., "Estimating Software Costs," Datamation, v 25 no 10, September 1979.

Racer, C.W., "Measuring Programming Productivity in the Maintenance Environment," Proceedings of SHARE 57, Chicago, Ill., 23-28 August 1981.

Scott, R.F., and Simmons, D.B., "Predicting Programming Group Productivity- A Communications Model," IEEE Transactions on Software Engineering, v se-1 no 4, December 1975.

Scott, R.F., and Simmons, D.B., "Programmer Productivity and the Delphi Technique," Datamation, May 1974.

Sholl, H.A., and Booth, T.L., "Software Performance Modeling Using Computational Structures," IEEE Transactions on Software Engineering, v se-1 no 4, December 1975.

Stevens, W.P., Myers, G.J., and Constatine, L.L., "Structured Design," IBM Systems Journal, v 13 no 2, 1974.

Wasserman, A.I., and Belady, L.A., "Software Engineering: The Turning Point," Computer, September 1978.

Wolverton, R.W., "The Cost of Developing Large-Scale Software," IEEE Transacitions on Computers, v c-23, no 6, June 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, Ca 93940	2
3.	Curricular Office, Code 37 Naval Postgraduate School Monterey, Ca 93940	1
4.	Assistant Professor Dan C. Boger, Code 54BK Administrative Science Department Naval Postgraduate School Monterey, Ca 93940	1
5.	Associate Professor Norm Lyons, Code 54LB Administrative Science Department Naval Postgraduate School Monterey, Ca 93940	1
6.	Chairman, Code 54 Department of Administrative Science Naval Postgraduate School Monterey, Ca 93940	1
7.	Fleet Material Support Office Code 92 Mechanicsburg, Pa 17055	1
8.	Fleet Material Support Office Code 92E Mechanicsburg, Pa 17055	1
9.	Fleet Material Support Office Code 92T Mechanicsburg, Pa 17055	1
10.	LCDR Gary J. Hughes, SC, USN Naval Supply Center Puget Sound Bremerton, Wa 98314	2

11. MAJ W. Helling 1
Commandant Marine Corps
Code MMOS
Headquarters Marine Corps
Washington, D.C. 20380